This page Is Inserted by IFW Operations
And is not part of the Official Record

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of The original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

**As rescanning documents *will not* correct images,
Please do not report the images to the
Image Problem Mailbox.**

THIS PAGE BLANK (USPTO)

(12) UK Patent Application (19) GB (11) 2 330 222 (13) A

(43) Date of A Publication 14.04.1999

(21) Application No 9721353.2

(22) Date of Filing 08.10.1997

(51) INT CL[6]
G06F 9/44

(52) UK CL (Edition Q)
G4A APL

(56) Documents Cited
EP 0803806 A2

(58) Field of Search
UK CL (Edition P) G4A APL
INT CL[6] G06F 9/44

(71) Applicant(s)
Mitel Corporation
(Incorporated in Canada - Ontario)
PO Box 13089, 350 Legget Drive, Kanata,
Ontario K2K 1X3, Canada

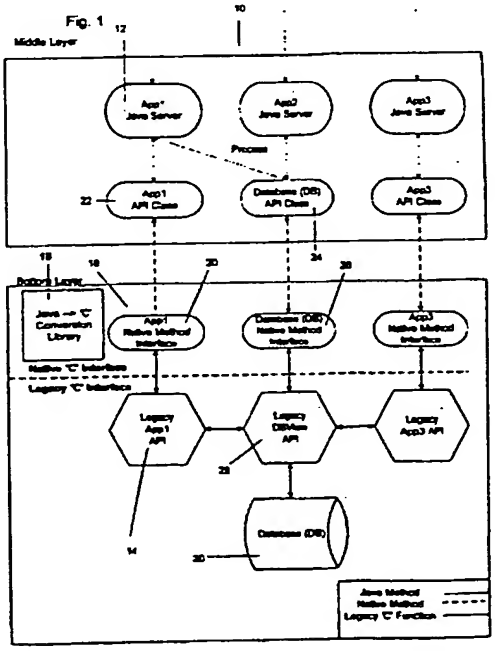
(72) Inventor(s)
David Simser

(74) Agent and/or Address for Service
Venner Shipley & Co
20 Little Britain, LONDON, EC1A 7DH,
United Kingdom

(54) Abstract Title
Bi-directional conversion library

(57) A bi-directional conversion library 18 is provided for translating data structures used in a computer program from a first computer programming language to data structures used by a second computer programming language. The method uses plural string, array functions for creating new string, array objects in the first and second languages based on string, array parameters passed to the functions from the second, first languages, respectively. Object/structure functions copy data between an object of the first language and a data structure of the the second language according to predetermined indicated class descriptors, and create new objects in the first language, based on one of either a default constructor or a specified data structure of the second language according to a specified class descriptor. Field accessor/mutator functions read and modify individual fields of an object in the two languages. In the preferred embodiment, a conversion library is provided for converting between Java and C. The conversion library allows existing computer programs to be reused, thereby reducing the effort required to integrate newly written programs to existing systems.

Fig. 1



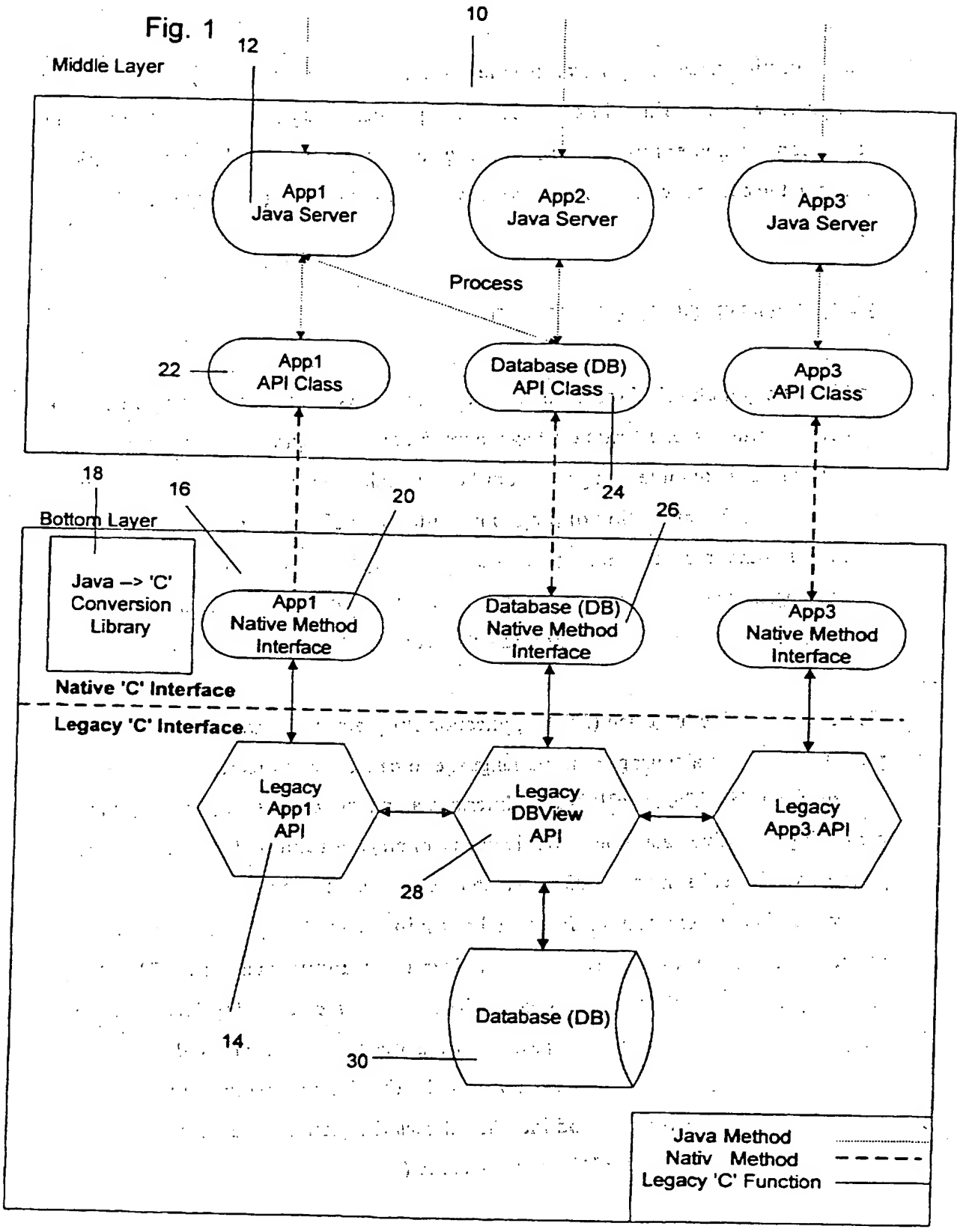
At least one drawing originally filed was informal and the print reproduced here is taken from a later filed formal copy.

This print takes account of replacement documents submitted after the date of filing to enable the application to comply with the formal requirements of the Patents Rules 1995

GB 2 330 222 A

BNSDOCID: <GB___2330222A_I_>

Fig. 1



BNSDOCID: <GB___2330222A__I_>

BI-DIRECTIONAL CONVERSION LIBRARY

FIELD OF THE INVENTION

This invention relates in general to a method for translating data structures used by a computer program written in a first computer programming language to its equivalent in a second computer programming language and more specifically for translating between the data structures of the two computer programming languages Java™ and C.

BACKGROUND OF THE INVENTION

In present day multi-platform computing environments, it is often a requirement that programs written in one language have to interact with programs written in another language (e.g. a computer program from Java™ making use of a C library). A prime difficulty with the interaction of programs written in different computer programming languages is the representation of data - each language stores information differently. If two programs are to interact, one program must convert the data from the representation used by the other program to its own.

The Java™ Native Interface (JNI) Application Programming Interface (API) is provided in the Java™ programming language to call C functions as if they were written in Java™. The Java™ Native Interface also provides a set of functions for converting primitive data types. While this is an elegant solution from a Java™ code perspective, it results in an interface which is extremely difficult to understand and to use from a C code perspective. It is well known in the art that typical computer programs use complex data structures built from the primitive data types. The JNI does not provide a method for automating the conversion of complex data structures. Also, complexity of this interface practically ensures that faults will find their way into applications where native methods are used. The danger presented by these faults is extreme since the native method interface allows C functions to access pointers to memory residing inside a Java™ virtual machine (VM).

The Java™ VM is intended to present an abstract, logical machine design free from the distraction of inconsequential details of any implementation. The memory areas of the Java™ virtual machine do not presuppose any particular locations in memory or locations with respect to one another. The memory areas need not consist of contiguous memory. However, the instruction set, registers, and memory areas are required to represent values of certain minimum logical widths (e.g. the Java™ stack is 32 bits wide). These requirements are discussed in the following sections.

The Java™ virtual machine is a machine which runs a special set of instructions which specify every possible instruction which may be implemented by the machine. The Java™ VM allows ever computer to run Java™ applications. A pointer is a data item which specifies a location of another data item. Pointers are very unstable and therefore their use can lead to unstable computer operations, which can lead to a possible crash of the entire Java™ VM.

To reduce the complexity of implementing native methods, and to minimize the number of places where pointers to memory within the Java™ VM are accessed, the inventors have recognized the desirability of providing a means to convert objects (or even primitive data types) used in legacy function calls (i.e. function calls provided by existing programs). In software, a legacy system is an existing system to which new code is being added and a legacy function is a function used in such a system. By localizing this conversion to a single location, much of the danger inherent in the use of native methods can be mitigated.

SUMMARY OF THE INVENTION

According to the present invention, a bi-directional conversion library is provided which translates and converts data structures used in a first programming language to equivalent data structures of a second programming language, and vice versa. In the preferred embodiment, using Java™ and C as the first and second programming languages, respectively, the library of the present invention utilizes the Java™ Native Interface (JNI). The JNI is the interface between Java™ and any other computer

programming language which is to be incorporated into Java™ applications during the conversion process. At the present time, only C and C++ are supported by the Java™ Native Interface. The bi-directional conversion library of the preferred embodiment may be implemented as an API provided by a Dynamic Link Library (DLL) for allowing Java™ data objects to access C functions via a mapping layer. Each DLL function takes a pointer to the JNI as a first parameter. By limiting the use of this pointer to a select set of functions, the problem inherent in the prior art Java™ Native Interface API of allowing C functions to access pointers to memory residing inside the Java™ VM is eliminated.

BRIEF DESCRIPTION OF THE DRAWINGS

A preferred embodiment of the invention is described below with reference to the sole drawing, in which:

Figure 1 is a schematic illustration showing the process architecture for layered interaction between Java™ servers and legacy C code according to the preferred embodiment.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

Figure 1 shows a preferred architecture for a Java™-to-C Conversion system **10**. In this architecture, Java™ data objects **12** access legacy C functions **14** via a mapping layer **16**. This layer **16** uses a bi-directional Java™-to-C conversion library **18** to convert Java™ data objects **12** into equivalent C structures **20** and calls an unmodified legacy C function **14**. Each legacy API **14** requires an associated mapping layer **16**, since mapping for each function may be different (e.g. a single Java™ data object **12** may map into multiple parameters in the function signature). The conversion library **18** provides a suite of functions to provide a consistent, well-defined approach to implementing the mapping layer **16**. Also although the disclosure hereinafter describes mainly the conversion from Java™ to C in detail, it will be understood by a person skilled in the art how to convert data structures from C to

BNSDOCID: <GB___2330222A__I_>

Java™, or with appropriate substitutions of functions, convert between two other different computer programming languages.

In the following description of Figure 1, a conversion from Java™ to C is disclosed using the preferred embodiment of the present invention. We will use a Java™ data object within App 1 12 as our focus for this description. The first action is that a method of the Java™ object App 1 12 is called by another Java™ process. After being called, depending on the call, it calls a method provided by its corresponding App 1 API class 22 or accesses a Database API class 24. After the App 1 API class 22 or the Database API class 24 has been accessed, the mapping or Native C layer 16 is accessed through a corresponding Native Method Interface, the APP 1 Native Method Interface 20 and a Database Native Method Interface 26 respectively. This means that the APP 1 API Class 22 calls the APP 1 Native Method Interface 20 and the Database API class 24 calls the Database Native Method Interface 26. The mapping layer 16 will take any Java™ data submitted from the calls from the API layer 22 located inside the middle layer and translate the Java™ data 12 into an equivalent C data structure. After translating the Java™ objects into equivalent C data structures, the Native Method Interface calls a corresponding legacy App 1 API 14. For the Database Native Method Interface 26, a call is made to a legacy Database view layer (DBView). If the legacy App 1 API 14 has been chosen, then it must access the legacy DBView API 28 since the legacy App 1 API requires the Database 30.

The conversion library 18 is an API provided DLL located within the Native C mapping layer 16 for linking a Java™ application to a C application by performing the necessary actions to translate Java™ data objects 12 to C data structures 14 and the complementary actions to translate C data structures 14 to Java™ data objects 12.

As discussed in greater detail below, the DLL implementing the conversion library 18 comprises four main groups of functions as follows: string functions, array functions, object/structure functions and field accessor/mutator functions. An accesor is a method or function that returns the value of encapsulated data while a mutator

BNSDOCID: <GB___2330222A__I_>

performs a complementary action setting a value of encapsulated data. It should be noted that every DLL function of the conversion library **18** takes a pointer to the Java™ Native Interface environment as a first parameter. A pointer is a data item which specifies the location of another data item. Limiting the use of this pointer to the conversion library, instead of allowing native methods to access the memory of the Java™ VM directly decreases the chance of crashing the Java™ VM , is discussed briefly above.

The group of string functions provides means for converting between Java™ String objects and a null terminated array of char representation used in C. Six functions are provided: `createJavaString`, `createJavaStringFromWide`, `allocCString`, `freeCString`, `allocCWideString`, and `freeCWideString`. It should be noted that the names of these functions and of the ones set forth below may be changed and still perform the same function provided that the changes are made globally (i.e, throughout the program). The names used in computer programs are mainly for the benefit of the programmer as a reminder of what each separate function or procedure does. Using the string functions as an example, the first function `createJavaString` is named so as to clearly indicate that it creates a Java™ String. Thus for example, if all of the occurrences of the phrase name `createJavaString` in the program were changed to `nothingcreated`, a call of the `nothingcreated` function would still perform the original function of creating a Java™ String object. Therefore, the conversion library is not dependent on the name of the functions and the choice of names is also not restricted.

The `createJavaString` function creates a new Java™ string object based on a C string passed as a parameter. A parameter is information located in another part of the computer program which is needed in order to run the calling function. In this case, the parameter is the C string which is to be translated to Java™ by the conversion library of the present invention. The Java™ string need not be freed, since Java's™ garbage collection is responsible for managing the memory allocated for the string object. Freeing of a string returns memory back to the computer allowing it to be

reused later. Java's™ garbage collection automatically determines when memory is no longer needed and frees the memory for later use. A representative API for the `createJavaString` function is as follows:

```
jstring createJavaString(
    JNIEnv * env,
  const char * cString );
```

The `createJavaStringFromWide` function creates a new Java™ string based on a wide C string passed as a parameter (i.e. a null-terminated array of unsigned short). This function is performed in a similar manner to the previously discussed `createJavaString` function and may be implemented by the following API:

```
jstring createJavaStringFromWide(
  JNIEnv * env,
  const unsigned short * cString );
```

The `allocCString` function creates a new C string based on a passed Java™ string object, and is the complement to the `createJavaString` function. The returned string is declared to be a constant (const) since the string is allocated within the Java™ VM and therefore should not be modified. This is also consistent with the treatment of strings within Java™ where a Java™ string object is immutable. A representative API for allocating is as follows:

```
const char * allocCString(
  JNIEnv * env,
  jstring javaStr );
```

Since there is no garbage collection in C, all strings created with `allocCString` must be freed via a call to `freeCString`, which may be implemented by the following API:

```
void freeCString(
```

```
    JNIEnv * env,
    jstring javaStr,
    const char ** stringToFree );
                                /* Note: pointer to a string */
```

The `allocCWideString` and `freeCWideString` functions perform analogous operations for wide C strings. The `freeCString` function takes a pointer to a string allocated via `allocCString` (const char **) as a parameter, not simply a string (const char *). This use of the pointer ensures that the freed string is not referenced a later time. Representative APIs for `allocCWideString` and `freeCWideString` are as follows:

```
const unsigned short * allocCWideString(
    JNIEnv * env,
    jstring javaStr );

void freeCWideString(
    JNIEnv * env,
    jstring javaStr,
    const unsigned short ** stringToFree );
                                /* Note: pointer to a string */
```

The array functions provide means for converting between Java™ arrays and their C counterparts where both arrays of primitive data types and of objects are supported. Primitive data types are defined as data elements built-in to the computer language which represent a single piece of data (eg. a number) . Examples of primitive data types include integers, float, and boolean. For the purposes of these functions, Java™ string objects are considered primitive data types, since a class description is not needed. Five functions are provided: `createJavaPrimitiveArray`, `createJavaObjectArray`, `allocCPrimitiveArray`, `allocCStructureArray` and `freeCArray`.

The `createJavaPrimitiveArray` and `createJavaObjectArray` functions both create a Java™ array based on a C array passed in as a parameter. In each case, a size and type of the array must also be specified. In the case of creating an object

BNSDOCID: <GB___2330222A__I_>

array, a class descriptor for an associated class must also be provided. In a similar manner to Java™ string objects, Java™ arrays need not be freed, as they are garbage collected by the Java™ programming language. Representative APIs for this function are as follows:

```
jarray createJavaPrimitiveArray(
  JNIEnv * env
  char * arrayType,
  void * cArray,
  int cArrayLen );

jarray createJavaObjectArray(
  JNIEnv * env
  void * cArray,
  int cArrayLen,
  classDesc_t * classDesc );
```

The `allocCPrimitiveArray` and `allocCStructureArray` functions both create a C array based on a Java™ array passed as a parameter. The returned array is a copy of the associated Java™ array so that any changes made to this new C array are not reflected in the original Java™ array. For both functions, the type of array must be specified. In contrast to the Java™ array creation functions, the C array creation functions return the length of the allocated array via a *numElements* parameter. The `allocCStructureArray` function also requires the class descriptor describing the Java™ array elements passed as a parameter. Exemplary APIs for `allocCPrimitiveArray` and `allocCStructureArray` are as follows:

```
void * allocCPrimitiveArray(
  JNIEnv * env
  jarray javaArray,
  char * arrayType,
  int * numElements );
```

```
void * allocCStrucutreArray(
  JNIEnv * env,
  jarray javaArray,
  char * arrayType,
  classDesc_t * classDesc,
  int * numElements );
```

As is also the case with the string functions, since C does not provide garbage collection, the `freeCArray` function must be called to release memory associated with the allocated C array. The `freeCArray` function may be implemented by the following API:

```
void freeCArray(
  JNIEnv * env,
  jarray javaArray,
  char * arrayType,
  void ** CArray );
```

The object/structure functions are at the core of the Bi-Directional Conversion Library according tot he present invention. These functions are used to convert between Java™ objects and C structures. Four object/structure functions are provided: `java2c`, `c2java`, `createEmptyJavaObject`, and `createJavaObjectFromStruct`.

The `java2c` function fills a specified C structure with data contained in a Java™ object passed to the function according to an indicated class descriptor. An API for implementing the `java2c` function is as follows:

```
void java2c(
  JNIEnv * env
  jobject javaObj,
  void * cStruct
  classDesc_t * classDesc );
```

BNSDOCID: <GB___2330222A__I_>

The `c2java` function performs the complementary action of copying the data within a C structure to a corresponding Java™ object according to an indicated class descriptor. A representative API for `c2java` is as follows:

```
void c2java(
  JNIEnv * env,
  jobject javaObj,
  void * cStruct,
  classDesc_t * classDesc );
```

It should be noted that the `java2c` and `c2java` functions of the preferred embodiment do not provide any facilities for converting static data members of a Java™ object.

The design of class descriptors is set forth herein below for use with the `java2c` and `c2java` functions of the conversion library according to the invention. The following description is predicated on the assumption that legacy C structures already exist, and that data in these structures is required in the Java™ programs or objects to be converted. The following steps illustrate the process of creating a Java™ class to parallel a C structure and describing the Java™ class via a class descriptor.

1) Identify the required C data structure (at the time of its creation, the library of the present invention assumes that legacy C structures already exist – otherwise, C and Java™ structures can be developed in tandem, as would be understood by a person of ordinary skill in the art).

2) Create a Java™ class with data members in a one-to-one correspondence with the C data structure identified in step 1 (even though this may not result in the best object-oriented design). This class must have a public default constructor defined. The default constructor is a constructor for a Java™ object which does not take in any parameters. It should also be noted that data members must not be class variables (i.e. not static data members). Any access specifiers can be used for data members (e.g. they can be private, protected or public).

BNSDOCID: <GB___2330222A__I_>

3) Define an array of field descriptors, with one entry per data member of the Java™ class created in Step 2. The following is a brief description of the information which must be defined:

- `fieldName`: name of a data member in the Java™ class
- `fieldType`: one of `JTYPE_*` as defined in `java2c.h`, discussed in greater detail below
- `fieldOffset`: an offset into the C structure of the associated field (use an "`offsetof`" macro provided by Microsoft's™ C compiler)
- `maxElems`:

  if `fieldType` is `JTYPE_STRING` or `JTYPE_WSTRING`, this is the maximum string length in characters

  if `fieldType` is `JTYPE_ARRAY` or `JTYPE_VARRAY`, this is the maximum number of elements in the array
- `arraySizeOffset`: if `fieldType` is `JTYPE_VARRAY`, this is the offset the field in the C structure which contains the actual number of elements in the array. This allows an array with a certain `maxElems` to be occupied by fewer elements, as specified in another field of the structure
- `fieldClassDesc` - if `fieldType` is `JTYPE_OBJECT ( className )`, `JTYPE_ARRAY ( JTYPE_OBJECT ( className ) )`, or `JTYPE_VARRAY ( JTYPE_OBJECT ( className ) )`, this is a pointer to the class for `className`

4) Define a class descriptor with the following information:

- `className`: the fully qualified Java™ class name using '/' as a class delimiter e.g. "`ops/dbview/myDbclass`"
- `numfields`: the number of fields in the field descriptor array
- `size`: the size of the assoicated 'C' structure (use the "`sizeof`" macro).
- `fields`: the array of field descriptors from 3)

BNSDOCID: <GB___2330222A__I_>

An actual example of the steps involved in creating the Java™ class to parallel a following hypothetical C structure `otherStruct_t` and its corresponding `otherClass` is disclosed herein below:

```
typedef struct    {
char someString [ MAX_STR_LEN + 1 ];
   int numValues;
   otherStruct_t values [ MAX_VALUES ];
} info_t;
```

The following Java™ class could be created (other methods can be added – only the data members matter for this library):

```
public class Info  {
   private String someString = null;
   private int numValues = 0;
   private otherClass [ ] values =
      new otherClass [ MAX_VALUES ];

   public Info()
   {
      // create otherClass objects in values array
};
   }
}
```

The following exemplary `fieldDesc_t` (field descriptor array) and `classDesc_t` (class descriptor array) definitions are defined:

```
fieldDesc_t infoFields [ ] = {
   { "someString",
     JTYPE_STRING,
     offsetof ( info_t, someString ),
     MAX_STR_LEN + 1,
     0,          // not applicable for this field type
     NULL    }, // not applicable for this field type
   { "numValues",
     JTYPE_INT,
```

BNSDOCID: <GB___2330222A__I_>

```
            offset ( info_t, numValues ),
            0,          // not applicable for this field type
            0,          // not applicable for this field type
            NULL    }, // not applicable for this field type
        {       "values",
            JTYPE_VARRAY (  JTYPE_OBJECT (  "otherClass"  )
    ),
            offsetof (  info_t,  values ),
            MAX_VALUES,
            offsetof (  info_t, numvalues ),
            &classDescForOtherClass }
    };
        classDesc_t infoClassDesc = {
            "Info",
            sizeof ( infofields ) / sizeof (  fieldDesc_t
    ), // num fields
            sizeof ( infor_t ),
            infoFields };
```

The `createEmptyJavaObject` function creates a new Java™ object of a specified class in a `className` parameter based on a default constructor. The data members of a newly created object are not assigned any values unless a default constructor performs initialization. A representative API for this function is as follows:

```
jobject createEmptyJavaObject(
  JNIEnv * env
  char * className );
```

The `createJavaObjectFromStruct` function creates a new Java™ object based on a specified class descriptor. The data members of the new Java™ object are initialized with the values contained in a C structure passed in as a `cStruct` parameter. The `createJavaObjectFromStruct` function may be implemented by the following API:

BNSDOCID: <GB___2330222A__I_>

```
jobject createJavaObj ctFromStruct(
  JNIEnv * env,
  void * cStruct,
  classDesc_t * classDesc );
```

It should be noted that both of the `createEmptyJavaObject` and `createJavaObjectFromStruct` functions each create a Java™ object using a default constructor. The default constructor must be defined for all objects to be created via the conversion library DLL of the present invention. If a constructor other than the default constructor is required, the JNI `NewObject` function must be chosen.

Often, not all of the data members of an object passed to a native method are required. The field accessor/mutator functions discussed below allow individual fields of an object to be read (via an accessor function) or modified (via a mutator function). None of these functions require a class descriptor to be defined. It should be noted that field accessor/mutator functions do not provide any facilities for accessing or changing static data members of a Java™ object.

Two sets of accessor/mutator pairs are provided in accordance with the preferred embodiment: `get<Primitve Type>Field/ set<Primitive Type>Field` and `getObjectField/setObjectField`.

The `get<Primitive Type>Field/set<Primitive Type>Field` pair of functions provide means to access and modify primitive data members of a Java™ object based on their names. Each primitive type has its own pair of accessor/mutator functions, each of which returns a value of the appropriate C type. The supported types are summarized below:

Field Type	<Primitive Type> in Function Name	Return/Parameter Type
JTYPE_INT	*Int*	int

BNSDOCID: <GB___2330222A__I_>

JTYPE_LONG	*Long*	_int64
JTYPE_SHORT	*Short*	short
JTYPE_BYTE	*Byte*	char
JTYPE_WCHAR	*Wchar*	unsigned short
JTYPE_CHAR	*Char*	unsigned char
JTYPE_FLOAT	*Float*	float
JTYPE_DOUBLE	*Double*	double
JTYPE_BOOLEAN	*Boolean*	boolean
JTYPE_STRING	*String*	jstring

The string type is a special case, since in Java™ it is a class, where as in C it is a primitive array. The bi-directional conversion library of the present invention treats the string as a primitive type (since a class descriptor is not required), but the Java™ string value returned cannot be used directly by the C code. The `allocCString` and `freeCString` functions must be used to allocate and free C-style strings. It should be noted that primitive data types are basic field types such as boolean , integer or byte.

The `get<Primitive Type>Field/set<Primitive Type>Field` functions may be implemented by the following exemplary APIs:

```
<Native Type> get <Primtive Type>Field(
        JNIEnv * env,
    jobject javaObj,
    char * fieldName );

void set<Primitve Type>Field(
    JNIEnv * env,
    jobject javaObj,
    char * fieldName,
    <Native Type> value );
```

BNSDOCID: <GB___2330222A__I_>

The `getObjectField` and `setObjectField` functions allow embedded objects to be accessed and modified. In addition to a field name in the Java™ object, the type of the object must also be specified. A `JTYPE_OBJECT (className)` macro can be used to formulate the type specifier. A skilled person will recognize that care should be taken when using the object accessor/mutator functions since the `getObjectField` function returns a reference to an actual embedded object, and not a copy of the object. The `setObjectField` replaces the reference to an existing embedded object with a reference to the new copied object. Typical usage of these functions would include using the `getObjectField` to retrieve a reference to an object, and then to use the `java2c, c2java` or a field accessor/mutator pair to manipulate the data. Since a reference to an original object is used, the `setObjectField` function need not be called (since it would merely change the reference back to the same object). If a new object is created via the `createEmptyJavaObject` or `createJavaObjectFromStruct` functions, the `setObjectField` function should be used to replace the existing embedded object with the new object. Exemplary APIs for the `getObjectField` and `setObjectField` functions are as follows:

```
jobject getObjectField(
   JNIEnv * env,
   jobject javaObj,
   char * fieldName,
   char * fieldType );

void setObjectField(
   JNIEnv * env,
   jobject javaObj,
   char * fieldName,
   char * fieldType,
     jobject value );
```

It will be appreciated that, although one particular embodiment of the invention has been described in detail, various changes and modifications may be made. For example, in the mapping layer, the conversion functions could be directly integrated

BNSDOCID: <GB___2330222A__I_>

into legacy C code. Also, although not provided by the implementation set forth herein, the mapping of primitive data types may be included in the conversion. Although the preferred embodiment provides for embedded objects to be embedded within themselves alternative embodiments are contemplated which do not support embedded objects or reference them to external structures. With respect to the mapping of arrays, embodiments are possible which do not to provide array mapping functions or array accessor/mutator functions. It is contemplated that access may be provided to Java™ Object Methods, as would be understood by a person of ordinary skill in the art. Furthermore, although the detailed description of the preferred embodiment herein above is limited to a discussion of the conversion between Java™ and C, it will be appreciated that conversions are possible between any two different computer languages according to the principles of the present invention, and in particular C can be converted into Java™. All such changes and modifications may be made without departing from the sphere and scope of the invention as defined by the claims appended hereto.

1. A bi-directional conversion library for converting data structures from a first programming language into data structures of a second programming language, comprising:

a) a plurality of string functions for creating new string objects in said first and second programming languages based on string parameters passed thereto from said second and first programming languages, respectively;

b) a plurality of array functions for creating new array objects in said first and second programming languages based on array parameters passed thereto from said second and first programming languages, respectively;

c) a plurality of object/structure functions for copying data between an object of said first programming language and a data structure of said second programming language according to predetermined indicated class descriptors and for creating new objects in said first programming language based on one of either a default constructor or a specified data structure of the second programming language according to a specified class descriptor; and

d) a plurality of field accessor/mutator functions for reading and modifying individual fields of an object in said first and second programming languages.

2. The bi-directional conversion library of claim 1, wherein said first programming language is Java™ and said second programming language is C, and wherein Java™ includes a Java™ Native Interface (JNI) facility.

3. The bi-directional conversion library of claim 2, wherein each of said functions takes a pointer to said Java™ Native Interface (JNI) environment as a first parameter.

BNSDOCID: <GB___2330222A__I_>

4. The bi-directional conversion library of claim 3, wherein one of said string functions creates a new Java™ string object based on a C string passed to said function as a parameter.

5. The bi-directional conversion library of claim 3, wherein one of said string functions creates a new Java™ string object based on a wide C string passed to said function as a parameter.

6. The bi-directional conversion library of claim 3, wherein one of said string functions creates a new C string based on a Java™ string object passed to said function as a parameter.

7. The bi-directional conversion library of claim 3, wherein one of said string functions creates a new wide C string based on a Java™ string object passed to said function as a parameter.

8. The bi-directional conversion library of claim 6 or 7, wherein one of said string functions frees said new C string for releasing memory allocated for said new C string.

9. The bi-directional conversion library of claim 3, wherein one of said array functions creates a new Java™ array based on a C array passed to said function as a parameter.

10. The bi-directional conversion library of claim 3, wherein one of said array functions creates a new C array based on a Java™ array passed to said function as a parameter.

11. The bi-directional conversion library of claim 10, wherein one of said array functions frees said new C array for releasing memory allocated for said new C array.

12. The bi-directional conversion library of claim 3, wherein one of said object/structure functions fills a predetermined C structure with data contained in a Java™ object passed to said function according to a predetermined one of said indicated class descriptors.

13. The bi-directional conversion library of claim 3, wherein one of said object/structure functions copies data within a predetermined C structure to a corresponding Java™ object according to a predetermined one of said indicated class descriptors.

14. The bi-directional conversion library of claim 3, wherein one of said object/structure functions creates a new Java™ object of a specified class in a className parameter based on a default constructor.

15. The bi-directional conversion library of claim 3, wherein one of said object/structure functions creates a new Java™ object based on a specified C data structure according to a specified class descriptor.

16. The bi-directional conversion library of claim 3, wherein one pair of said field accessor/mutator functions provides access to and modification of primitive data members of a Java™ object based on names of said data members.

17. The bi-directional conversion library of claim 3, wherein one pair of said field accessor/mutator functions provides access to and modification of embedded Java™ objects.

18. An Application Programming Interface (API) comprising the bi-directional conversion library of claim 1 implemented as a Dynamic Library Link (DLL) for linking a first application written in said first object-oriented language with a second application written in said second object-oriented language.

The Patent Office

Application No: GB 9721353.2
Claims searched: 1

Examiner: Leslie Middleton
Date of search: 30 March 1998

Patents Act 1977
Search Report under Section 17

Databases searched:

UK Patent Office collections, including GB, EP, WO & US patent specifications, in:

UK Cl (Ed.P): G4 (APL)

Int Cl (Ed.6): G06F 9/44

Other: Online: WPI, INSPEC, COMPUTER

Documents considered to be relevant:

Category	Identity of document and relevant passage	Relevant to claims
A	EP0803806 A2 (International Computers Limited) See whole document	

21

X	Document indicating lack of novelty or inventive step	A	Document indicating technological background and/or state of the art.
Y	Document indicating lack of inventive step if combined with one or more other documents of same category.	P	Document published on or after the declared priority date but before the filing date of this invention.
&	Member of the same patent family	E	Patent document published on or after, but with priority date earlier than, the filing date of this application.

BNSDOCID: <GB___2330222A__I_>

THIS PAGE BLANK (USPTO)